JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 391–422 (1999)

Practice

Evolution in Software Product Lines: Two Cases

MIKAEL SVAHNBERG* and JAN BOSCH
Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, S-372 25 Ronneby, Sweden

SUMMARY

This paper discuss the results of two case studies from a technical perspective, concentrating on the evolution of software assets in two Swedish organizations that have employed a product-line architecture approach for several years. This paper describes and analyses the commonalities and differences of these two cases, emphasising categories of the evolution of the requirements, of the software architecture and of the software components. This paper concludes with three types of lessons learned about evolution in software product lines: three evolution categories are predominant, three other categories are less significant but still common, and seven guidelines for software product-line evolution emerge. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: software product lines; software evolution; case study; object oriented frameworks

1. INTRODUCTION

The wide-scale reuse of software has been a long standing ambition of the software engineering industry (McIlroy, 1969; Parnas, 1976). The cost and quality of much of the currently developed software is far from satisfactory. In the situation where a software development organization markets a family of products with overlapping, but not identical functionality, the notion of a software product-line is a feasible approach to decrease cost and increase the quality of software (Linden, 1998; Bass *et al.*, 1997; Bass *et al.*, 1998; Dikel *et al.*, 1997; Macala, Stuckey and Gross, 1997). The software product-line defines a software architecture shared by the products and a set of reusable components that, combined, make up a considerable part of the functionality of the products. Especially in the Swedish industry, this is a logical development, since software is an increasingly large part of products that traditionally were considered to be primarily mechanical or electronic. In addition, software often defines the competitive advantage. When moving from a minor to a major part of products, the required effort for software development also becomes a major issue. This motivates industry to explore alternatives to increase the reuse of existing software to minimize product-specific development and to increase the quality of software.

A number of authors have reported on industrial experiences with product-line architectures, primarily from large American software companies, often defence-related. Bass *et al.* (1997,1998)

*Correspondence to: Mikael Svahnberg, University of Karlskrona/Ronneby, S-372 25 Ronneby, Sweden.
Email: Mikael.Svahnberg@ipd.hk-r.se

CCC 1040–550X/99/060391–32$17.50

Received 4 July 1999
Revised 23 September 1999

report results from two workshops on product-line architectures. Also, Macala *et al.* (1997) and Dikel *et al.* (1997) describe experiences from using product-line architectures in an industrial context. In Europe considerable effort has also been invested in investigating software product lines. Most notably, the 4th framework EU projects ARES (Linden, 1998) and PRAISE involve several major European industrial companies.

Software product lines present an important approach to increasing software reuse and reducing development cost by sharing an architecture and a set of reusable components among a family of products. However, evolution in software product lines is more complex than in traditional software development since new, possibly conflicting, requirements may originate from the evolution of existing products in the product line and the incorporation of new products.

Although the notion of software products lines has been studied by several authors, the primary effort has been directed to the conversion towards and the initiation of a software product line. Consequently, its evolution has been studied considerably less. In this paper, we address this by presenting the results of two case studies of software product line evolution. Two software assets were studied:

- an object-oriented framework for file systems, which is one of the primary components in the software product line of Axis Communications AB, producing a wide range of printer, storage, scanner and camera server products worldwide, and
- the Billing Gateway software product line, one of the flagship product families of Ericsson Software Technology.

We have studied the evolution of each software asset over several versions—eight and four releases respectively—and investigated what changes were made to the requirements, the software architecture, and the components and their implementation. Based on these cases, we discuss the commonalities and differences, presented as categories of evolution for each of these aspects, i.e., requirements, architecture and components.

The remainder of this paper is organized as follows. In the next section, the notion of software product line evolution is discussed in more detail and definitions for some concepts are presented. Section 3 discusses the research method that was used and its concrete application to this study. The cases are presented in Sections 4 and 5, respectively. The cases are compared and discussed in Section 6. The paper is concluded in Section 7.

2. SOFTWARE PRODUCT-LINE EVOLUTION

2.1. Context for evolution

Like nearly all pieces of software, a software product line evolves over time. The evolution can be viewed from an organizational and from a process perspective. The evolution of a software product line is driven by changes in the requirements on the products in the family. These new and changed requirements originate from a number of sources, such as the customers using the products, future needs predicted by the company, and the introduction of new products into the product line.

Figure 1 presents an overview of the evolution of a product line. Each business unit is responsible for one product or a small set of highly related products, and is interested in supporting a set of requirements. These requirements are divided into the requirements specific to the product that

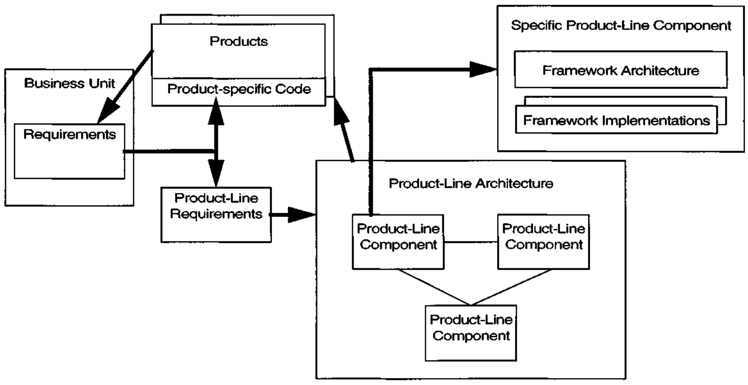

Figure 1. Evolution of a product line

this business unit is responsible for, and the requirements common for all or most products in the software product line. The requirements on a particular product are naturally implemented in the product-specific code, but the common requirements are supported by the software product line architecture and component set. Changes to the latter category, i.e. the product-line requirements, can affect the software architecture of the product line and the architecture of a particular component. This may create a change in its interface, or it can cause a change in one or more of the concrete implementations of the component. In some cases, the requirements may even cause a split of one component into two or more components, or the introduction of a completely new component into the product-line architecture.

The software product-line architecture and its component set are subsequently used to construct new releases or versions of the products in the family. These new versions of the products lead to new requirements, which are fed back into the business unit requirements, thus completing the cycle.

2.2. Our view of products, architectures and software product lines

We define a software product line as consisting of a software product-line architecture, a set of reusable components and a number of software products. The products can be organized in many ways. One example is the hierarchical organization presented in Figure 2, with the architectural variations organized accordingly.

We define a software product-line architecture as a structure comprised of software components, connectors establishing relationships among the components, and the externally visible properties of those components (Bass, Clements and Kazman, 1998). The role of the software product-line architecture is to organize the commonalities and variabilities of the products contained in the software product line and, as such, to provide a common overall structure.

A component in the architecture implements a particular domain of functionality; for example, the file system domain or the network communication domain. Components in software product lines, in our experience, are often implemented as object-oriented frameworks. Thus, a product is constructed by composing the frameworks that represent the component in the architecture.

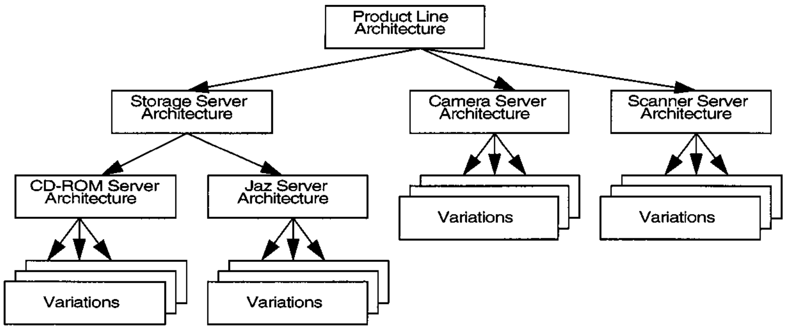

Figure 2. A hierarchic view of a generic product line, adapted from Bosch (1999b)

As shown in Figure 1, we define a framework to consist of a framework architecture and one or more concrete framework implementations. As an example, the file system framework component used by Axis Communications has an abstract architecture where the conceptual entities and their relations are identified. The framework implementations are the specific implementations for each concrete file system to be implemented, such as FAT, UFS and ISO-9660. Another example is the communications component in the Billing Gateway case discussed below, where there is a common API, or framework interface, that is used by all implementations of network protocols such as TCP/IP and X.25. This interpretation of a framework holds up well in a comparison with the findings of Roberts and Johnson (1996). Those findings included a white box framework that can be mapped to our framework architecture, and a black box framework that consists of several of our framework implementations in addition to the framework architecture.

The products in the software product line are instantiations of the product-line architecture and of the components in the architecture. There may also be product-specific component extensions and, generally, product-specific code is present to capture the product-specific requirements.

3. CASE STUDY METHOD

The main goal in our study was to find out how a framework evolves when it is part of a software product line. To achieve this goal, we conducted interviews with key personnel that have been involved in the studied systems for quite some time. Furthermore, we studied requirement specifications, design documents, and functional specifications that we got either from the companies' intranets, or from other sources. Some of the acronyms we encountered are summarized in Table 1.

Before the interviews, much time was spent on studying the available documentation in order to get a good background understanding. The interviews were open, and consisted mostly of encouragements to the interviewed people to help them remember how and in what order things happened. The interview findings were correlated with what was found in the documentation, and the interviewees were later asked some further questions, to resolve discrepancies and clarify things that were unclear.

The two case sites were selected in a non-random manner. The first case, at Axis Communications, was selected because Axis is a large software company in the region. Axis is

Table 1. Acronyms used in case studies descriptions

Acronym	Expansion and explanatory comment
ARES	Architectural Reasoning for Embedded Software, a research project sponsored by the European Union
ECOOP	European Conference on Object-Oriented Programming
FAT	File Allocation Table, a disk file system used by Microsoft Corporation
FODA	Feature-Oriented Domain Analysis, an analysis method developed at Carnegie Mellon University
FTP	File Transfer Protocol, a standard for transmission of data files
KLOC	Thousands of lines of code, a crude measure of software size
MUPP	Multi-User Parallel Proprietary file system at Axis Communications
PCNFS	Personal Computer version of the NFS, a file system
PRAISE	Product-line Realisation and Assessment in Industrial Settings, a research project sponsored by the European Union
NASA	National Aeronautics and Space Administration, a USA government agency
NDS	Novell Directory Service, a network management protocol from Novell
NFS	Network File System, used by Sun Microsystems in its Solaris system
NOSAR	Nordic Network on Software Architecture Research, a research network
RAID	Redundant Array of Independent Disks, a disk storage system
RISE	Research in Software Engineering, a research group at the University of Karlskrona/Ronneby
RSEB	Reuse-Driven Software Engineering Business, a phrase used by Ivar Jacobson, Martin Griss, and Patrik Jonsson
SCSI	Small Computer Systems Interface, a hardware standard for peripherals
SMB	Server Message Block, part of a network file system used by Microsoft
SNMP	Simple Network Management Protocol
UFS	Unix File System, a file system used by Sun Microsystems

part of a government-sponsored research project on software architectures, of which our University is one of the other partners, and Axis has a philosophy of openness, making it relatively easy to conduct studies within this company. The second case, at Ericsson Software Technology, was chosen because its close-to-campus site facilitates interaction with them, and because our University has had a number of fruitful cooperations with Ericsson in the past. Despite this ‘convenience sampling’, we believe that the companies are representative of a larger category of software organizations. The companies are presented in further detail below.

4. CASE 1: STORAGE SERVER

4.1. The company

In this case study at Axis Communications AB we studied the evolution of a major component in the architecture of the product line, an object-oriented framework for file systems. We describe two generations of this component, consisting of four releases each.

Axis Communications is a relatively large Swedish software and hardware company that develops networked equipment. Starting from a single product, an IBM print-server, the product line has now

grown to include a wide variety of products such as camera servers, scanner servers, CD-ROM servers, Jaz servers, and other storage servers.

Axis communications has been using a product-line approach since the beginning of the 1990s. Their software product line consists of reusable assets in the form of object-oriented frameworks. Currently, the assets are formed by a set of 13 object-oriented frameworks, although the size of the frameworks differs considerably.

The layout of the software product line is a hierarchy of specialized products, of which some are product lines in themselves. At the topmost level are elements that are common to all products in the product line. Below this level are assets that are common to all products in a certain product group. Examples of product groups are the storage servers, including for example the CD-ROM server and the Jaz server, and the camera servers. Under each product group there are a number of products, and below this are the variations of each product. Figure 2 illustrates the layout of the software product line and the variations under each product group.

Each product group is maintained by a business unit that is responsible for the products that come out of the product line branch, and also for the evolution and maintenance of the frameworks that are used in the products. Business units may perform maintenance or evolution on a software asset that belongs to another business unit, but this must be done with the consent of the business unit in charge of the asset.

The company uses a company-standard view on how to work with their software product line, even if their terminology might not be the same as that used by academia. A product-line architecture, in the Axis view, consists of components and relations. The components are frameworks that contain the component functionality. We feel that this is a beneficial usage of the framework notion, since each component covers a particular domain and can be of considerable size, e.g., up to 100 KLOC. However, compared to the traditional view (Roberts and Johnson, 1996), frameworks are usually used in isolation, i.e., a product or system uses only one framework. In the Axis case, a framework consists of an abstract architecture and one or more concrete implementations. Instantiation of the framework is thus made by creating a concrete implementation by inheriting from the abstract classes, and plugging this into place in the product. Further information on how the product line is maintained, and the issues concerning it, have been reported by Bosch (1999a, 1999b).

4.2. The product line studied

During this study we focused on a particular product, the storage server. This is a product initially designed to plug CD-ROM devices into the network. This initial product later evolved into several products, of which one is still a CD-ROM server; there is now also a Jaz server and a HardDisk server was recently added to the collection of storage servers. Central to all these products is a file system framework that allows uniform access to all the supported types of storage devices. In addition to being used in the storage servers, the file system framework is also used in most other products, since these products generally include a virtual file system for configuration, and for accessing the hardware device that the product provides network support for. However, since most of the development on the file system framework naturally comes from the storage business unit, this unit is the one responsible for it.

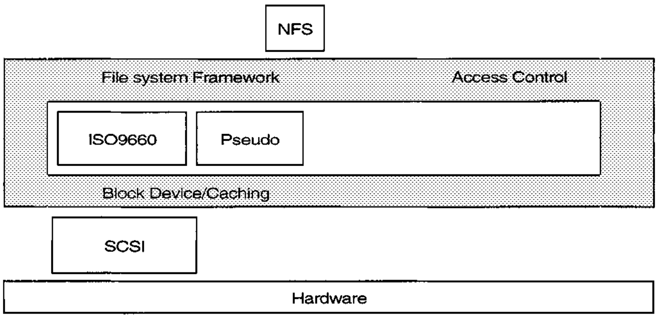

Figure 3. Generation 1 of the file system

The file system framework has existed in two distinct generations, of which the first was only intended to be included in the CD-ROM server, and was hence developed as a read-only file system. The first generation consists of a framework interface, under which there are concrete implementations of various file systems like ISO-9660 (for CD-ROM disks), Pseudo (for virtual files), UFS (for Solaris-disks), and FAT (for MS-DOS and MS-Windows disks). These concrete file system implementations interface to a block device unit, which in turn interfaces with a SCSI bus. Parallel with the abstract framework is an access control framework. On top of the framework interface, various network protocols are added such as NFS (UNIX), SMB (MS-Windows), and Novell Netware.

This is illustrated in Figure 3, where it is shown that the file system framework consists of both the framework interface, and the access control that interfaces with various network protocols like NFS. We also see that the block device unit and the caching to support device drivers such as the SCSI unit are part of the file system framework, and that the concrete implementations of file system protocols are thus surrounded by the file system framework.

The second generation was similar to the first generation, but the modularity of the system was improved. Also, it was developed to support read and write file systems, and was implemented accordingly. In addition, the framework was prepared for incorporating some likely future enhancements based on the experience from the first generation. Like the first generation, the first release of this framework only had support for NFS, with SMB being added relatively soon. An experimental i-node-based file system was developed, and was later replaced by a FAT-16 file system implementation.

Figure 4 depicts the second generation file system framework. As can be seen, the framework is split into several smaller and more specialized frameworks. Notably, the Access Control part has been 'promoted' into a separate framework.

4.3. Generations and releases

4.3.1. Generation 2, release 1

We now present the major releases of each generation in further detail, and focus on a particular product line component, the file system framework, and its concrete implementations.

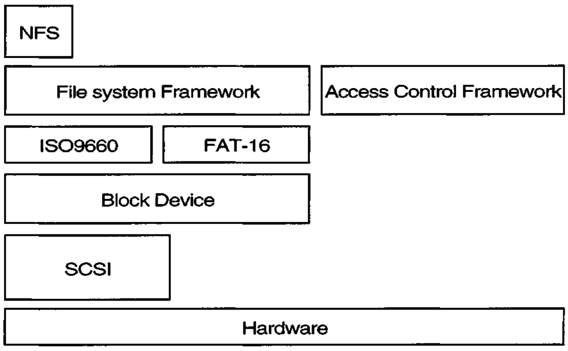

Figure 4. Generation 2 of the file system

The requirements on this, the very first release of the product, were to develop a CD-ROM server, a device that distributes the contents of a CD-ROM over the network. This implies support for network communication, network file system support, CD-ROM file system support, and access to the CD-ROM hardware. The birth of the CD-ROM product started this product line. The product line later branched into three different sub-trees with the introduction of camera servers and scanner servers.

Requirements on the file system framework were to support the CD-ROM file system, i.e., the ISO-9660 file system, and a virtual-file pseudo file system for control and configuration purposes. In release 1, it was only required that the framework supported the NFS protocol.

In Figure 3 we see a subsection of the product-line architecture where the NFS protocol, which is an example of a network file system protocol, interfaces with the file system framework. The file system framework in turn accesses a hardware interface, in this case a SCSI interface. Not shown in Figure 3 is how the network file system protocol connects to the network protocol components to communicate, using for example TCP/IP, and how these protocol stacks in turn connect to the network hardware interfaces.

The two framework implementations, the ISO-9660 and the Pseudo-file system implementations differed of course in how they worked, but the most interesting difference was in how access rights were handled differently in the two subsystems. In the ISO-9660 implementation, access rights were handed out on a per-volume basis, whereas in the Pseudo-file system, access rights could be assigned to each file. This was done with the use of the two classes, NFSUser and NFSAccessRight.

Later in the project, SMB was added as well, which did not change much since SMB is not that different from NFS. Code was added to handle access control for SMB, specifically to handle the classes SMBUser and SMBAccessRight.

In retrospect, it is hard to agree upon a reason for why the code was bound to a particular network file system, i.e., NFS. Inadequate analysis of future requirements on the file system framework could be one reason. Shortage of time could be another reason; there might not have been enough time to do it flexibly enough to support future needs.

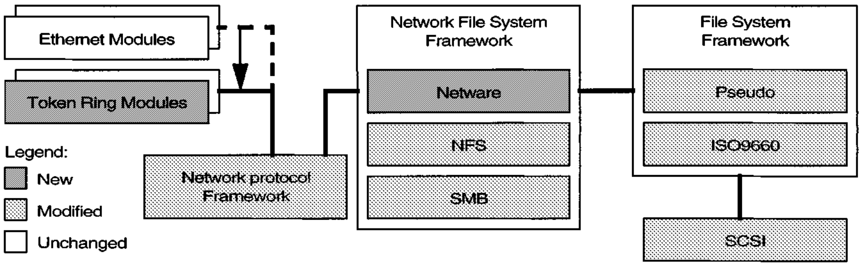

Figure 5. Changes in generation 1, release 2

4.3.2. Generation 1, release 2

The goal for the second product line release was, among other things, to create a new product. To this end, a shift from Ethernet to Token Ring was conducted. In addition, support for the Netware network file system was added, plus some extensions to the SMB protocol. Furthermore, the SCSI module was redesigned and support for multisession CDs was added, as well as a number of other minor changes.

The effect on the product-line architecture was that Netware was added as a concrete implementation in the network file system framework, modules for Token Ring were added, and a number of framework implementations were changed, as discussed below. Figure 5 summarizes how the product line architecture was changed by the new product requirements.

The addition of the Netware protocol could have had severe implications on the file system framework, but the requirements were reformulated to avoid these problems. Instead of putting into the framework the requirements that were known to be hard to implement in the existing architecture, the specification of the Netware protocol was changed to suit what could be done within the specified time frame for the project. This meant that all that happened to the file system framework was the addition of the NWUser and NWAccessRight classes.

Basically, it was this access-rights model that could not support the Netware model for setting and viewing access rights. The NWUser and NWAccessRight classes were added to the implementations of the Pseudo-file system and the ISO-9660 module. Furthermore, the ISO-9660 module was modified to support multisession CDs.

Late in this project a discrepancy was found between how Netware handles file-IDs compared to NFS and SMB. Instead of getting to the bottom of the problem and fixing this in the ISO-9660 module, a filename cache was added to the Netware module. Again, the file system framework interface managed to survive without a change.

4.3.3. Generation 1, release 3

Release 3 of the file system framework was primarily concerned with improving the structure of the framework and with fixing bugs. The SCSI driver was modified to work with a new version of the hardware, and a web interface was incorporated to browse details mainly in the pseudo-file system that contains all the configuration details. Support for long file names was also implemented, as was support for PCNFS clients.

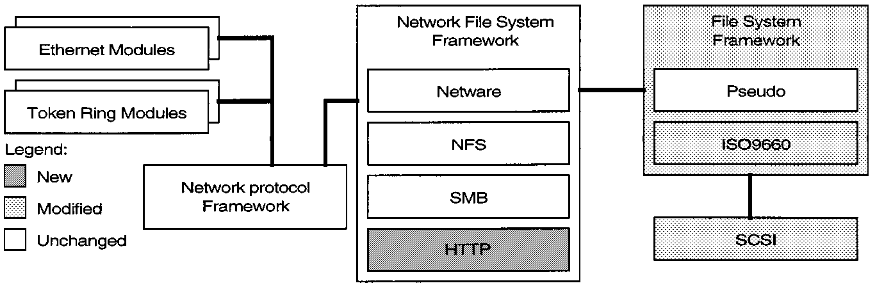

Figure 6. Changes in generation 1, release 3

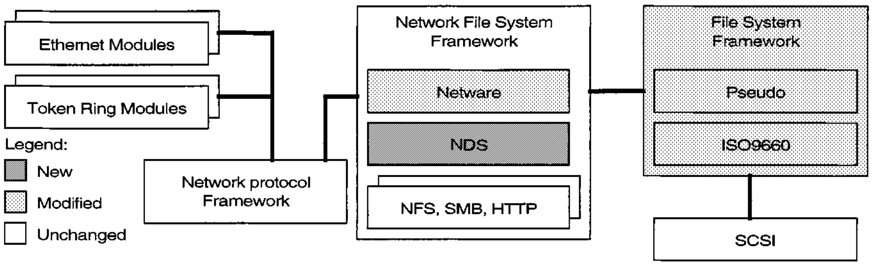

Figure 7. Changes in generation 1, release 4

The effects on the product-line architecture were, consequently, small. A new framework implementation was added to the network file system framework to support the web-interface. Figure 6 shows the modifications in release 3.

On the component level, the SCSI driver was modified to better utilize the new version of the hardware that supported more of the SCSI functionality on-chip instead of in software. Long filename support was added in the ISO-9660 module. This change had an impact on the framework interface as well, since the DirectoryRecord class had to support the long filenames. This implies that the interface of the file system component changed, but this merely caused a recompilation of the dependent parts, and no other changes.

4.3.4. Generation 1, release 4

This release was to be the last one in this generation of the file system framework, and was used as part of the new versions of the two aforementioned CD-ROM products until they switched to the second generation two years later. Requirements for this release were to support NDS, which is another Netware protocol, and to generally improve the support for the Netware protocol. Figure 7 summarizes the changes in release 4.

As in the previous release, the product-line architecture did not change in this project but, as we will see, several framework implementations were changed to support the requirements. NDS was added as a new module in the network file system framework, and was the cause for many of the other changes presented below.

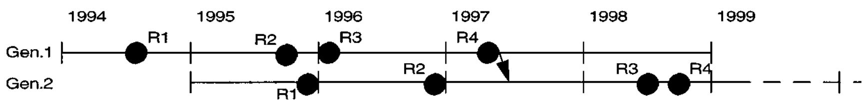

Figure 8. Timeline of the file system framework

Adding NDS provided the file system framework with a totally new way of acquiring the access rights for a file. Instead of having the access rights for a file inside the file only, the access right to the file is calculated by hierarchically adding the access rights for all the directories in the path down to the file to the access rights that are unique for the file itself. This change of algorithm was implemented in the access control part of the file system. Unfortunately, the access control parts for the other network file system protocols had to be modified as well, to keep a unified interface to them.

The namespace cache that was introduced in release 2 was removed, and the requirement was now implemented in the right place, namely in the ISO-9660 subsystem. This meant that the namespace cache could be removed from the Netware module.

4.3.5. Generation 2, release 1

Work on generation 2 of the file system framework was done in parallel to generation 1 for quite some time. As can be seen in Figure 8, the two generations existed in parallel for approximately four years. However, after the fourth release of generation 1, all resources, in particular the staff, were transferred into the development of the second generation, so parallel-development was only conducted for around two years. The transfer of resources is signified by the arrow between generations 1 and 2 in the picture.

Requirements on release 1 of this second generation were very much the same as those on release 1 of the first generation, with the exception that one now aimed at making a generic read–write file system from the start; it was realized at an early stage that the previous generation was unfit for this. The experiences from generation 1 were put into use here and, as can be seen when comparing Figure 4 with Figure 3, the first release of the second generation was much more modularized. As before, only NFS and SMB were to be supported in this first release, but with both read and write functionality. A proprietary file system, jocularly termed MUPP in Swedish, was developed in order to gain insight into the basics of an i-node-based file system.

Some framework implementations could be reused from generation 1, and modified to support read and write functionality. An example of this is the SMB protocol implementation.

4.3.6. Generation 2, release 2

Release 2 came under the same project frame as the first release, since the project scope was to keep on developing until there was a product to show. This product, a Jaz-drive server, went into major release 2 of the file system framework. Because of the unusual project plan, it is hard to find out what the actual requirements for this particular development step were, but it resulted in the addition of a new framework implementation in the file system, the FAT-16 subsystem, and the removal of the MUPP file system developed for release 1. The NFS protocol was removed,

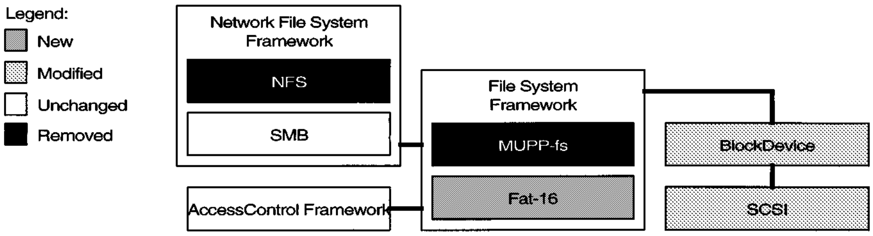

Figure 9. Changes in generation 2, release 2

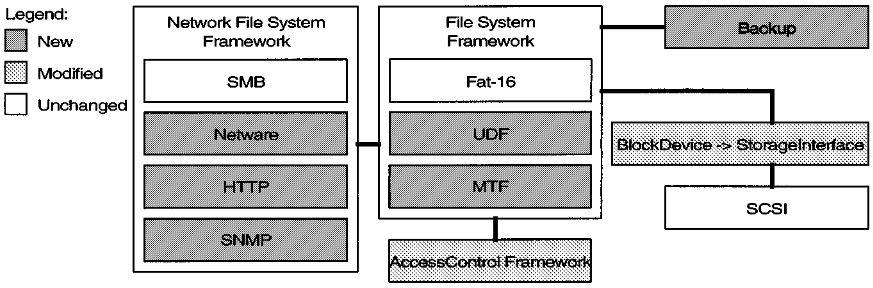

Figure 10. Changes in generation 2, release 3

leaving the product only supporting SMB. Some changes were made in the SCSI module and the BlockDevice module.

Figure 9 illustrates the changes conducted for this release on the product-line architecture. As can be seen, the actual file system framework architecture remained unchanged in this release, mainly because the volatile parts have been broken out to separate frameworks in this generation. As identified among the general deltas of this release, the implementation of FAT-16 was added to the collection of concrete file system implementations, and the MUPP file system was removed.

The product intended to use this release was a Jaz server, and this is what caused the changes to the SCSI and the BlockDevice module. Axis found that Jaz drives use an uncommon SCSI dialect, and the SCSI driver needed to be adjusted to use this. The BlockDevice module changed to use support the new commands in the SCSI driver.

4.3.7. Generation 2, release 3

The project requirements for release 3 were to develop a hard disk server with backup and RAID support. In order to support the backup tape station, a new file system implementation had to be added. Furthermore, it was decided to include support for the i-node-based file system UDF, as well as the FAT-16 system from the previous release. Figure 10 depicts the changes that took place in release 3.

The backup was supposed to be done from the SCSI-connected hard disks to a likewise SCSI-connected backup tape station. The file system for use on the tapes was MTF. SMB and Netware were the supported network file systems. Moreover, the product was to support an additional

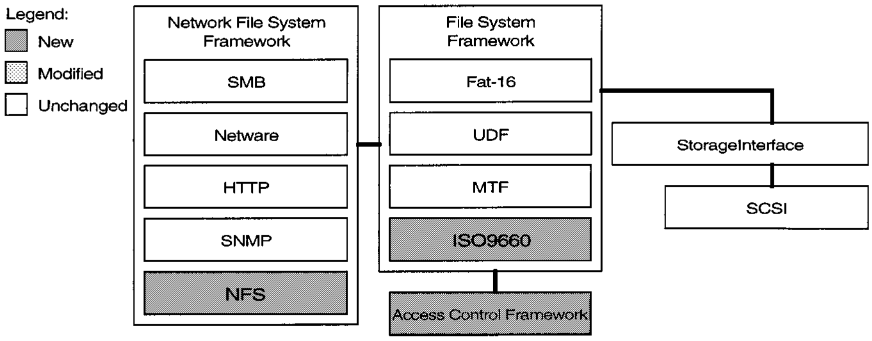

Figure 11. Changes in generation 2, release 4

network management protocol, called SNMP. RAID support was delegated to RAID controllers on the SCSI bus.

Once again, the overall product-line architecture remained more or less unchanged for this release. Most of the changes happened inside the product-line components, with the addition of new framework implementations. The only major change was the addition of the Backup component, which was connected to the file system framework. Furthermore, the BlockDevice changed its name to StorageInterface, and adding Netware caused modifications in the access control framework.

This was the first time that Netware was supported in generation 2, and it reused components from generation 1 and modified them to support write functionality. Likewise, the web interface was reused from generation 1. The access control framework was modified to work with the new network file systems, Netware, HTTP and SNMP.

4.3.8. Generation 2, release 4

The product requirements for this release were to make a CD server working with a CD changer. This is the first CD product developed using generation 2 of the file system framework, and the requirements were inherited from the last CD product developed using generation 1, and the last project using the second generation. Some minor adjustments were made to these requirements; one group of requirements dealt with how CDs should be locked and unlocked; another group concerned the physical and logical disk formats; and a third group concerned caching on various levels.

The product-line architecture again survived relatively unchanged by all these requirements. An implementation of ISO-9660, supporting two ways of handling long filenames (Rockridge and Joliet) was developed under the file system component, and the NFS protocol was re-introduced to the set of network file systems. As we can see, there are, once again, additions of framework implementations, but the architecture does not change in itself. Figure 11 depicts the changes made in release 4.

For this release, the AccessControl framework was not only modified to work with Netware, it was completely rewritten and the old version discarded. The version existing in release 2 was a quick fix that could not support the required functionality.

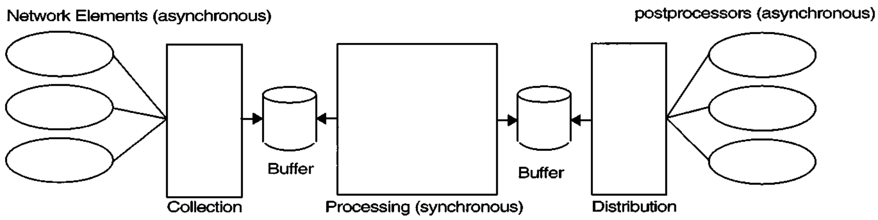

Figure 12. Top-level architecture of BGw

5. CASE 2: BILLING GATEWAY SYSTEM

5.1. The company

The second case study was Ericsson Software Technology, in particular the software product line for the Billing Gateway system. This section presents the company, the system, and the four available releases of the system.

Ericsson Software Technology is a leading software company within the telecommunications industry. The company is located in southern Sweden and employs 800 people. Our study was conducted at the site in Ronneby, which is closest to our University. This section of the company has been developing a software product line for controlling and handling billing information from telecommunication switching stations since the beginning of this decade. Whereas the software product line in Axis Communications is centered around products for the consumer market, the product line in Ericsson Software Technology is focused on developing products for large customers, so there is typically one customer for each product released from the software product line. Mattsson and Bosch (1998, 1999a, 1999b) discuss the company and the Billing Gateway product further.

5.2. The system

The Billing Gateway (BGw for short) is a mediating device between telephone switching stations and various postprocessing systems such as billing systems, fraud control systems, etc. The task of the Billing Gateway is to collect call data from the switching stations, process the data in various ways, and distribute it to the postprocessing systems. The processing that is done inside the BGw includes formatting (e.g., to reshape the data to suit the postprocessing systems), filtering (e.g., to filter out the relevant records), splitting (e.g., to clone a record to be able to send it to more than one destination), encoding and decoding (e.g., to read and write call data records in various formats), and routing (e.g., to send records to the appropriate postprocessing systems).

One of the main roles of the system architecture was to buffer the transfer of data between controlled processing units handling synchronous data, and the network elements and the post-processing systems handling asynchronous data. To achieve this, the system was divided into three major parts: collection, processing and distribution, as can be seen in Figure 12. Inside the processing node, there are four separate components handling encoding, processing, decoding and data abstraction, as shown in Figure 13.

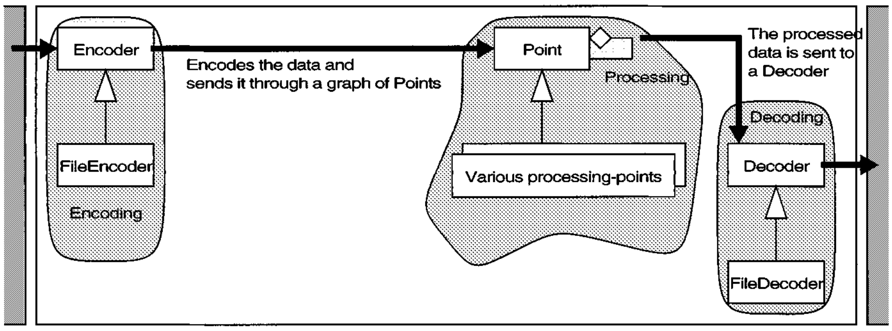

Figure 13. Architecture of the BGw processing node

Collection and distribution is solved using a Communication API (CAPI). This CAPI is file-based, using the primitives `connect`, `transfer`, `commit`, `close` and `abort`. The call data is put into the file buffer, where it is picked up by the processing node once the node is ready to accept a new file.

Inside the processing node, the files are first encoded into an internal format, which is then sent through a network of processing points, such as formatters and filters, that process the data in some way. Last in the network, the data are coded into an output format and then put into the outgoing buffer for further distribution to post-processing systems.

As opposed to the Storage Server case, Ericsson Software Technology uses the traditional framework view, in that the entire Billing Gateway product is considered a framework that is instantiated for each customer. According to Mattsson and Bosch (1999b), over 30 instantiations of the Billing Gateway have been delivered to customers. However, to view the Billing Gateway as one framework yields, in our view, unnecessary complexity, and we find it is better to consider the components in the product, and to regard these from a framework perspective, as we did in the Storage Server case. If we adjust our view accordingly, we see that the product is instantiated in much the same way as the Storage Server. In this, concrete implementations are inherited from abstract interfaces, where they are part of the running product instantiated according to each product's configuration.

5.3. BGw releases

5.3.1. Release 1

The BGw has had four different releases, each of which is described below. For each release, we present the requirements on the system and the impact these requirements had on the architecture. Some of the terms used are specific to the telecommunications domain, and unless they are of relevance to this paper they will not be explained further.

Ericsson Software Technology had previously developed several systems in the Billing Gateway domain, and was interested in creating a software product-line to capitalize on the commonalities between these systems. At the next opportunity, when a customer ordered such a system, they took the chance and generalized this system into the first release of the Billing Gateway. The

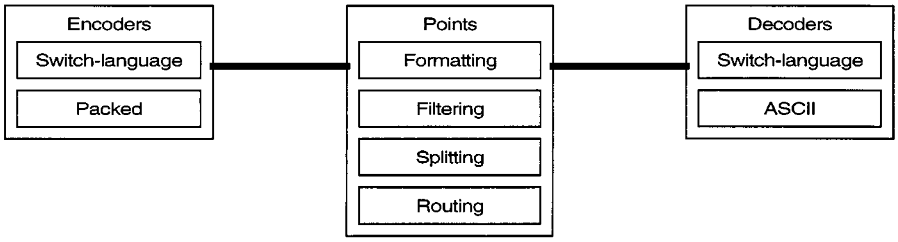

Figure 14. BGw release 1

requirements on this release were, as described above, to collect call data from network elements, process this data, and distribute the data to post-processing systems. Two types of network elements were supported. Communication to these was done using X.25, and using a switch language known by the switching-stations, or in other forms that some switches use, for example a packed version of the switch language. Moreover, the BGw was required to support an interpreted language used in the processing nodes to perform conversions and other functions on the call data. Distribution of the processed data was done over NFS, FTP, or to a local disk. In addition to this, a number of requirements were aimed at improving fault tolerance, performance and scalability.

Since this release is a first, the entire product line architecture is new, as are the implemented components. Figure 13 illustrates the overall architecture inside the processing node, and Figure 14 illustrates what instantiations of the components in the architecture were made. The protocols that were implemented were chosen from a pool of protocols planned for implementation. As is presented below, the remaining protocols were implemented in release 2.

When developing release 1, few concrete thoughts were dedicated towards the requirements that might appear in the future. Rather, the focus was on making as good a design as possible to accommodate the current requirements, using sound design principles to create a generic, flexible and reusable system.

5.3.2. Release 2

Release 2 can be seen as the conclusion of release 1, and the requirements were invented or found by the developers, covering what they thought was of importance. In release 2, the maintenance team completed the generalizations begun in release 1. The concepts from release 1 were cultivated further and most of the work took place inside the product. Some new network protocols were added, and the data transfer was done using memory instead of files, as was the case in release 1. Most notably, the inheritance hierarchies in release 1 had been very deep, and these were now broken up into aggregate relationships instead. This was done because it was realized that it is easier for the team to add classes to an aggregate relationship than to a class hierarchy. Figure 15 illustrates how the inheritance hierarchies were transformed. In effect, what was done was to apply the state and strategy patterns (Gamma *et al.* 1995).

According to the maintainers interviewed, what was done in release 2 was merely to conclude release 1 by implementing the rest of what was already planned, including network protocols and class generalizations. In this sense, they said, release 2 would be better named release 1.1.

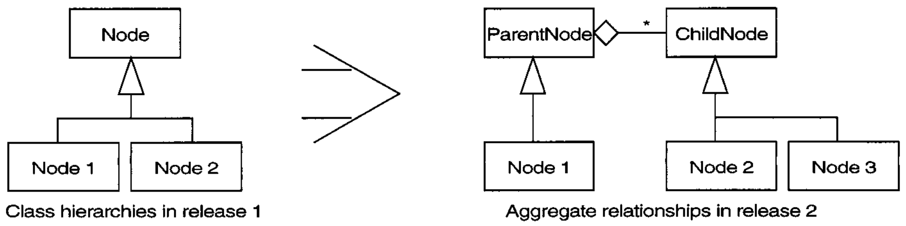

Figure 15. Break-up of the class hierarchies

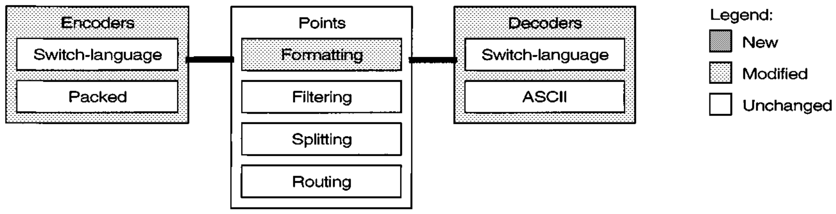

Figure 16. BGw release 2

Consequently, the effects this release had on the product-line architecture were relatively small. In addition to the continued work of generalizing, a number of new encoders and decoders were supported. Furthermore, moving to memory-based data transfer changed the encoders by providing a MemoryEncoder in addition to the FileEncoder.

Figure 16 illustrates the changes in release 2. As can be seen, the encoding and decoding frameworks change because of the change to memory-based transfer, and the formatting unit is extended by supporting new default formatters. In fact, the formatting unit did not change because new default formatters were added, since these were implemented in the BGw internal language, and are interpreted during run-time.

5.3.3. Release 3

When trying to ship release 2 to the market, it was realized that the product did not quite satisfy the potential customers' needs. Small things were missing, which made it harder to compete in the market. This led to release 3, where the demands of the market were implemented. Specifically, the interpreted language used in the processing nodes was too simple for the needs of many customers, and the file-based transaction was too coarse. The changes in this release were thus to introduce a new, more powerful language, and to shift from file-based transactions to block-based.

The introduction of a new language affected the point-nodes, since these use the language module. The introduction of the new language went fairly painlessly, and the impact was of a relatively local scope. However, moving to a block-based transaction scheme turned out to be more troublesome. The entire system had been built for processing large files only seldomly, e.g., a 4 MB file once in four minutes, and the new block-based transaction assumed that the system would process a large number of small files fairly often, e.g., eight 2 KB blocks per second. A new Communication API

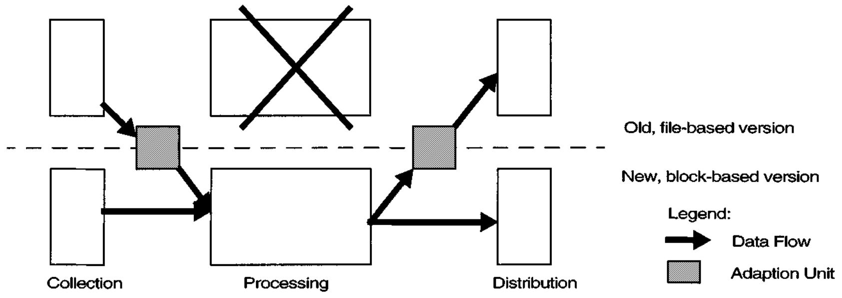

Figure 17. Introduction of block-based transactions

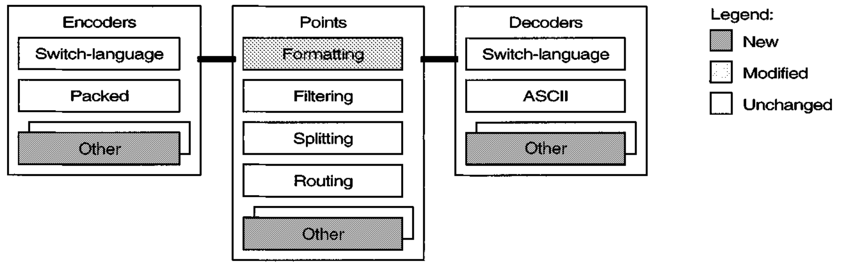

Figure 18. BGw release 3

(CAPI), a Block CAPI, was developed, as was a new processing node. The old CAPI could still be used for data of the old type, but had to be adapted to the new system.

Figure 17 depicts the top-level changes that were made to the product-line architecture in release 3. The new nodes, the collection and distribution nodes and the processing node, were constructed by scavenging the old code. Figure 18 summarizes the changes in release 3. In addition to the larger changes, new network protocols were supported, and some new point-nodes were added. As in release 2, new default formatters were added and new encoders and decoders were implemented. On the distribution side, some new network protocols were also added.

5.3.4. Release 4

Almost as soon as release 3 was completed, work started on defining the requirements for release 4. For various reasons this release is referred to as BGwR7. What had happened since release 3 was that the customers had run the BGw application for a while, and now required additional functionality. The major changes were that the BGw was ported from a Sun Microsystems (Sun) platform to a Hewlett–Packard (HP) platform, a database was added as a post-processing system, the set of encoders and decoders were made dynamically configurable, and verifications were made to ensure that all the data were received from the network elements and that nothing was lost during the transfer. In addition, some new point-nodes were implemented, a number of new library functions were added to the interpreted language in the formatters, and lookup tables for

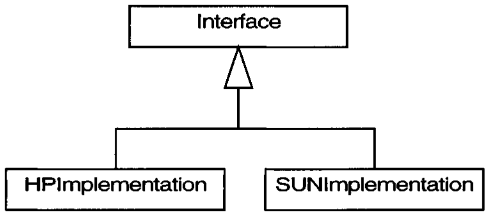

Figure 19. Example of adding classes for HP support

temporary storage of data in tables inside the BGw were added. Furthermore, some new network standards and data formats were accommodated.

Architecture-wise, a module for the lookup tables was added, and the addition of a database as a post-processing system caused some design changes. The database was supposed to store the call data in an unformatted and uncached way, which meant that the steps in the BGw where the data records are decoded to a post-processing format and written to disk for further distribution had to be skipped in the database case. Instead, the data had to be sent directly from the penultimate point in the point-graph to the database, since the last point in the graph codes the data to an output format.

Apart from this, the product-line architecture remained the same. Inside the components, more things happened. HP handles communication differently to Sun, so the port to HP led to new classes in many class hierarchies, as is illustrated in Figure 19. To dynamically configure the set of coders did in fact not cause as much trouble as would be expected. The functionality already existed in another component, namely the one loading library functions for the interpreted language, and this software could be reused for the coders as well. The requirements on verification that all data were received forced changes in all the in-points in the point-graph, since this is the first place where the data can be examined inside the BGw. The remaining changes—the new points, the library functions, the lookup table, and the new standards—did not cause any trouble, and were implemented by creating sub-classes in inheritance hierarchies, and by plugging in functions according to existing guidelines.

Figure 20 illustrates the changes made in release 4. The changes caused by the port to HP are left out for clarity. Also, several new encoding formats were supported in release 4, but as in release 3 these are quite technical, and we believe that they are not relevant to our focus here on product-line architecture.

6. DISCUSSION

6.1. Comparison complications

If we study the differences between the two cases, we see that the terminology used in the two companies is different. Whereas Ericsson Software Technology considers the entire Billing Gateway to be a framework, Axis Communications talks about frameworks on the subsystem level. Just two cases are, of course, not enough to assign significant importance to this difference, but at Ericsson Software Technology the BGw is the flagship of the company, whereas at Axis the Storage Server

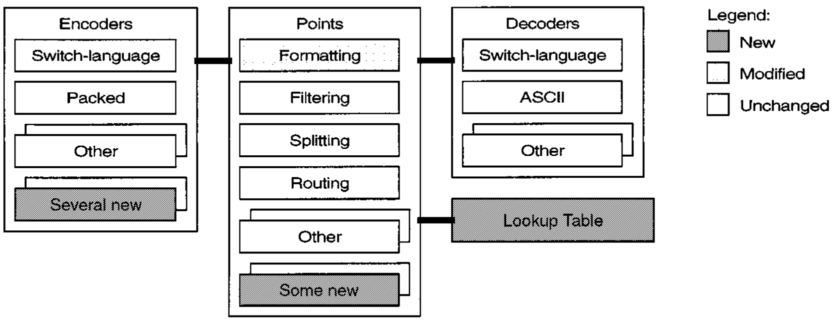

Figure 20. BGw release 4

is merely one product among many others, and the actual flagship of Axis is the 13 reusable assets from which the products are built. This could be what causes the difference in terminology.

Another difference is how Axis allows experimentation, in that software can be developed for the sole purpose of learning. Consider, for example, the in-house developed MUPP file system that was present in the system for one product release, and then removed and replaced with other file systems. We also see that networked file system protocols like NFS have been developed, put into one or more products, and then removed for some reason. Ericsson, on the other hand, operates in a much more goal-determined manner. Everything that is put into the product stays, whether it is used or not.

6.2. Common categories

When examining the evolution of the two cases, we see that the requirements, the product-line architecture components, and the evolution of the product-line architecture all neatly fall into a set of categories that are common to both cases. Figure 21 presents these categories in summary form, and points out the relations between the sections. We discuss below the categories, from the perspective of the requirements categories.

- **New product family.** When a new product family is introduced into the software product line, the maintainer is faced with the decision of whether to clone an existing product-line architecture (split the product line), or to create a branch (derive a product line) in the software product line. At Axis, both have happened, whereas in the Ericsson case branching is preferred. New products and product families generally imply that some components are added, changed or removed. Indirectly, this also leads to new or changed relations between the components.
- **Introduction of a new product.** A new product differs from the previous members in the product family by supporting more functionality, or the same functionality in a different way. This is achieved either by adding a new component or by changing an existing component in some way. In some cases, the changes required are so vast that it is more cost-effective to simply rewrite the component from scratch to replace the existing component, drawing from

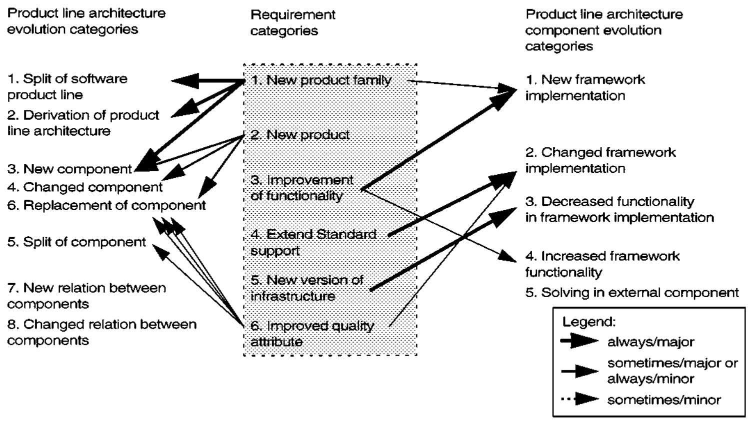

Figure 21. Relationships among the categories of product-line evolution

the experience gained from the previous version, but designing to incorporate the new desired functionality.

- **Improvement of functionality.** This is the common evolution direction both at Axis and at Ericsson, where the driving requirement is to keep up with the plethora of new standards as they become available. Commonly, what this leads to is a new implementation of an existing framework architecture. The architecture generally does not change, nor do the other framework implementations, even if the framework interface sometimes needs to be adjusted. This is also the evolution presented by Roberts and Johnson (1996), where concrete implementations are added to the library of implementations.
- **Extension of support of standards.** For various reasons in both cases studied, a decision was made not to support an industry standard fully in an initial release of the software, but rather to support initially a subset of a standard. In subsequent product releases, the support was extended until the entire standard was supported. Extending the support for a standard most often involved changing an existing framework implementation, in order to incorporate the additional functionality. An example of this is the SMB networked file system protocol in the Storage Server that has been implemented in turns.
- **New version of infrastructure.** As Lehman (1994) states, functionality has a tendency to move from the perimeter of a system towards the centre, as the innermost layers extend and support more new functionality. This category is more characteristic of the Axis case because of the in-house developed CPU, for which new versions are frequently released. Since the Billing Gateway runs on a standard operating system, the Billing Gateway system experiences fewer changes in the infrastructure. What this leads to normally is that the functionality implemented in a certain framework implementation decreases.
- **Improvement of quality attribute.** At Axis, the quality attribute to improve is mainly maintainability, whereas at Ericsson, most of the efforts are directed at performance. However, it is hard to say what effect changing the quality attribute will have on the architecture or the components in the software product line, because every type of quality attribute, as described by McCall (1994), will have a unique set of relations to the product-line

architecture and the product-line architecture components. However, most of the effects are covered by the arrows in Figure 21—i.e., improvement of a quality attribute results in new, changed or replaced components. Specifically, for example, a changed component implies in the Ericsson case that a particular framework implementation has been changed to better meet performance requirements. Maintainability requirements can in some cases be met by splitting a component, in order to break out functionality that no longer fits into the original component description.

- **External component insertion to add functionality yet minimize framework architectural impact.** This is an interesting category, which we have found evidence of not only in our cases, but also in other companies. As a project begins to draw towards its end, there is a growing reluctance to alter the underlying frameworks, since this yields more tests, and a higher risk of introducing bugs that have an impact on a large part of the system. A new requirement is then added (or an existing requirement is reinterpreted), leading to changes with architectural impact, if implemented normally. To avoid the excessive amount of effort required to incorporate the architectural changes, the maintainer instead develops a solution that implements the requirement outside the framework. This has the benefit of localizing possible problems to a particular module, as well as avoiding the need to modify all of the calling components late in the project. However, being the 'hack' it is, adding a component outside the framework violates the conceptual integrity (Brooks, 1995) of the component architecture.

6.3. Component impacts

In both of our cases, we see that at some point in time a large section of the product is thrown away and replaced by a newly-implemented section. This section replaces the old one completely, and does not add any really revolutionary functionality. In the Axis case, it was to support write-functionality in the file system framework, and in the Ericsson case, the transaction model did not work together with the block-based data units. In both cases, it is a single new requirement on the product that forces the developers to discard a whole section and rewrite it.

Also in both cases, it was decided that instead of trying to adjust an old component to work with the new requirements, it was better to start from scratch and salvage whatever code could be reused in the new component. We also see that it is not the code in itself that is the problem, nor is it the components, the architecture or the overall concepts. The system still works under the same general 'work description', and it still uses the same nominal entities to accomplish its work. The problem is more subtle.

Why some requirements have such an impact on one or a set of components is not quite clear. We believe the reason is that some design decisions and assumptions do not take on an explicit representation in the final architecture and, consequently, become embedded imperceptibly in the source code. Active design decisions are thus reformed into implicit assumptions in the source code. When such a design decision alters because of some new or changed requirement, there is no easy way to find all the places where this design decision is used, and it is hence easier, or believed to be easier, to rewrite the component from scratch.

A clear example of this can be seen in release 3 of the Billing Gateway case. There, the 2K-block-based transaction model was introduced. The BGw thus changes from a batch-sequential system to a

pipes-and-filters system (Shaw and Garlan, 1996), meaning that data could flow through the system instead of being buffered between every processing station. The original design decision to process large files relatively seldomly was hence changed to process small files often. In the first model there were no definite time constraints on how long it should take to process one data record, whereas in the second model, the data records could not be buffered and processed as a batch job.

A clear example can be also seen in the Storage Server case. There, the requirement to support write as well as read functionality led to the new generation of the file system framework. While reading is a very passive thing in most file systems (Unix file systems may require an update on the 'last read' variable), writing to a file system is, on the other hand, a completely different matter. Files and directories can change, giving rise to many new requirements; for example, on caches and access control. Rather than patch the first generation, a new version was created that could handle all the new requirements. Here we see that the initial design decision to support only read functionality changed, and that write functionality could not be solved parallel to the previous functionality, since the read functionality depends on certain aspects of the write functionality. An initial decision to support only read functionality was thus implicitly assumed as each line of code was written, and any attempts to code for a future change in this proved insufficient.

6.4. Related work

Jacobson, Griss and Jonsson (1997) address many issues regarding reuse, but in particular discuss application family engineering, component system engineering and application system engineering. Covered are the requirements elicited from use-cases, robustness analysis, design and implementation, and also testing of the application family. However, the evolution of the architecture and the components is addressed only marginally. Reuse is assumed to be out-of-box, and the particular problems that arise when other products depend on the evolution of a common asset are, as far as we understand, not covered. Our work does not discuss the aforementioned topics that are covered by Jacobson, Griss and Jonsson. Instead we study the area of evolution from a technical perspective, and the guidelines presented are more focused towards the later stages (traditionally called maintenance or post-delivery) of the life cycle.

The Product Line Practice initiative from the Software Engineering Institute (SEI) resulted in a number of workshops on the topic of product-line architectures and the development of these. The reports from these workshops provide much insight into the state of the art regarding product lines. The first workshop (Bass *et al.*, 1997) focused very much on charting the situation—how the participants of the workshop worked with their product line with regard to a number of topics. The second workshop (Bass *et al.*, 1998) focused more on giving a general insight into some topics that the previous workshop had identified. Both of these reports are very broad, and do not provide any in-depth study of the topics. Positioning our work with relation to this, we focus on the technical aspects of evolution, and provide more detailed analyses of evolution compared to the wide but shallow surveys in the two SEI workshop reports.

Another similar initiative is ARES, which focuses on methods, techniques and tools for supporting the development of software for embedded systems in product families. The end goal is to find ways to design software for software product lines, to help design reliable systems with various quality attributes that also evolve gracefully. See Linden (1998) for further information on this ESPRIT project.

Dikel *et al.* (1997) conducted a study at Nortel examining the success factors for Nortel's newly created product-line architecture. They identified six factors which they considered critical and evaluated the product-line architecture with respect to them: focusing on simplification, adapting to future needs, establishing an architectural rhythm, partnering with stakeholders, maintaining a vision, and managing risk and opportunities. For each of these, a rationale is presented, and an example of how the company (Nortel) works with respect to that factor. The product lines in our study are, we believe, quite similar to that of Nortel, and the traps described have also been encountered by Axis and Ericsson. The focus of the Nortel study differs from ours, but we believe they complement each other, the Nortel study from a management perspective and ours from a technical perspective.

Macala, Stuckey and Gross (1997) report on product-line development in the domain of air vehicle training systems. In this project, four key elements were enforced: the process, the domain-specificity, technology support, and the architecture. From the experience gained, seven lessons and 13 recommendations are given, which are mostly concerned with project management. Most of these lessons and recommendations are not specific to software product-line thinking, and there is little emphasis in the paper on the actual software product line. This study, compared to that of Dikel *et al.* (1997), is on a more detailed level, presenting guidelines for the micro-processes, whereas Dikel *et al.* presents more top-level management advice. Similar to the Dikel case, we see that this study is not focused on predicting changes, other than that they recommend having at least a five-year plan for the products upon their insertion into the product line.

Extending the scope, Basili *et al.* (1996) present a case study done at NASA to build up an estimation model for predicting maintenance effort. Whereas this study is not done on a product-line architecture, the data presented covers the activities in maintenance project life cycles, which is in our experience not so different from that of a product-line project. In both cases there is an existing code base which must be considered, and the project phases are also almost the same (Chapin, 1988). The study covers as many as 25 releases, and measures things like effort per activity (analysis, isolation, design, code and test, and inspection) and effort per type of change (adaptive, corrective, perfective). The study assumes that the different types of change are evenly spread, meaning that including all changes, from costly to cheap, the effort estimations hold, but the model cannot be used to effectively predict the impact of a particular change request. The taxonomy presented in our work provides a more detailed view of the evolution, and the model from Basili *et al.* can be used per requirement category, and hence gives more accurate estimations per change request.

Another example of an evolution paper is that of Perry and Siy (1998), which describes an approach for managing parallel evolution in a large product, the 5ESS switching system from Lucent Technologies. This study provides valuable insight regarding configuration management. In the Axis case, the two generations of the file system framework coexisted for some years, which makes some of the configuration management issues relevant, even if the two companies in our study do not have as rigorous an approach to parallel development as Lucent Technologies was reported to have.

Lehman (1980) presents a set of laws of evolution. While at a high level, they are nevertheless applicable to our study as well. In particular, the laws of continuing change and increasing complexity are extremely important to consider during development using a software product line. If we relate this to our work, we see that our work provides better insight into the dynamics of Lehman's laws by describing the evolution and providing some basic steps one can take to prevent deterioration.

When it comes to categorizing evolution, Mattsson and Bosch (1998) describe how frameworks evolve, and the types of changes that a framework can become subject to. They state that a framework can undergo (a) internal reorganization, (b) a change in functionality, (c) an extension of functionality, and (d) a reduction of functionality. These categories of change are also found and confirmed in our study.

Lindvall and Runesson (1998) present data from a case study regarding changes between two versions of a system. This system, as far as we know, is not part of a product line. From the presented figures we deduce that the evolution has been markedly different from that of our study, in that many more classes have been changed rather than, as in our case, added. However, we still find some evidence of our guideline regarding general component interfaces. In their study, 44.4% of the changes are visible, but the union of these changes and internal class body changes consists of as much as 79.4% of the changes, which indicates that many changes can be kept out of the interfaces. However, it should be noted that the figures from their study are on a class level and not on a component or framework level, as in our study.

A topic that is related to software product lines is domain-specific software architectures, as for example, that reported by Tracz (1995). A domain specific software architecture is a set of components that are specialized for a particular type of task (domain). Thus, the relation to a software product line is that a product-line architecture component contains a domain-specific software architecture. The Tracz (1995) report is particularly interesting in our case, since it also discusses requirements, especially reference requirements, which are requirements that drive the design of the abstract domain-specific software architecture.

FODA (Kang, 1990) is a method for domain analysis. Part of FODA is concerned with constructing graphs of features, in order to spot commonalities and benefit from these commonalities when planning reuse and adding new features. Much effort is put into the analysis model to foresee future enhancements on the features or components. Comparing FODA to our work, we see that FODA focuses on the spotting of commonalities by constructing domain-specific feature categories, and the relations between the categories. The categories we have presented are not based on particular features, but rather on possible steps of evolution. In that sense, FODA is orthogonal to our work. Related to FODA is FeatureRSEB (Griss, Favaro and d'Alessandro, 1998), which extends the use-case modeling of RSEB (Jacobson, Griss and Jonsson, 1997) with the feature model of FODA.

The Billing Gateway has been studied in earlier publications by our research group. The papers most closely related to our subject are those by Mattsson and Bosch (1999a, 1999b). Mattsson and Bosch (1999b) present metrics data from the four Billing Gateway releases. The data include the number of classes, the number of added classes per release, and change rates for four of the major subsystems. Based on these data, modules likely for restructuring are identified using an adapted version of a prediction approach proposed by Gall *et al.* (1997). The other paper (Mattsson and Bosch, 1999a) presents another set of metrics on the releases of the Billing Gateway, where the purpose is to replicate a case study by Bansiya (1998, 1999) that presents an approach to assess framework maturity using a set of design metrics. Whereas the main focus of these two papers may not be exactly in line with our research, the data presented for the four releases of the Billing Gateway gives context for the data reported in this paper.

The Axis Communications systems have not as yet been researched as much as Ericsson's Billing Gateway systems. There are two initial studies conducted on the Axis product line and the

development process, and reported by Bosch (1999a, 1999b). These papers present an overall view of the architecture and the development work. Problems are identified and discussed, and a number of research issues are presented.

7. CONCLUSIONS

7.1. Lessons learned

From the two product-line cases studied, we learned that three categories of evolution are predominant, discussed below in Section 7.2. We believe that these categories of evolution are also predominant in the general case. Furthermore, we learned that three other evolution categories occur more frequently than others. These we discuss below in Section 7.3. Figure 21 shows the relationships we found among the categories of evolution. Lastly, we learned that five guidelines apply to the development and evolution of software product lines. We discuss them below in Section 7.4.

In this paper we presented two cases of product-line evolution. One was of common evolution, namely that software product lines evolve by the addition of new component implementations of the existing component interfaces without touching these interfaces. The other was of less common but still frequent evolution, where the component implementations are modified for various reasons. In addition, two frequent occurrences of what, at a quick glance, seems to be evolution categories, have been shown to be more concerned with the initiation of the software product line than the actual evolution of it. In our opinion, these findings are strong enough to be considered further when designing and maintaining a software product line, and focus design and work efforts accordingly.

7.2. Common evolution

The three categories of evolution, described earlier in Sections 6.2 and 6.3 and Figure 21, that occur much more frequently than others are:

- improvement of functionality, often to support market and industry standards changes;
- changed component to support product change; and
- new framework implementation related to infrastructure change.

These are the most common ways that a product line evolves, namely improving on the functionality of the components by supporting new industry standards. Granted, these three categories of evolution do not cost nearly as much as when the architecture changes as a consequence of the evolution. Still, we believe that the data are strong enough to suggest that it is very important to spend time in the design phase to ensure that future functionality fits into the existing interfaces, so that architectural change can be minimized.

The data are also strong enough to support the case that if requirements of these three categories are detected in the requirements specification, a fairly accurate estimation of the impact of these requirement can be made, which helps in planning a project. Furthermore, by examining earlier requirements on a software product line, one can better estimate the current state of erosion that the product line is in, by examining how many requirements the product line has been subjected to from these three categories.

7.3. Other evolution

In addition to the common categories of evolution presented above, three other evolution categories (see Figure 21) are worth mentioning further because of their high rate of occurrence. These are: introducing a new component, replacing a component, and changing a framework implementation.

- **Changed framework implementation.** During the development of the Storage Server, it happened frequently that only part of a standard was implemented in the first release, and that additional parts were added in later projects. Together, these implementations changed the framework implementation. Likewise, in the Billing Gateway, the work done for release 2 was mostly concerned with restructuring the object-oriented design, which caused changes of a structural nature to many of the components. The conclusion is that it is an ongoing activity to restructure the software in order to prevent premature aging, and to extend the support for various industry standards in order to better compete on the market.
- **New component.** The single requirement to add a new product to the product line, or to start a new product line, injects a large number of new components into the product-line architecture. However, once a product is ensconced in the product line, very few new components are added. Examples are the first releases of the Storage Server and the Billing Gateway. Since this is where the software studied started, all components are naturally new. While this new component category occurs fairly frequently in our cases, we have seen that the occurrences are mostly concerned with the creation of the first version of the software product line. This is, we argue, also true in the general case: that to add new components to the architecture after the initial version is created is not done very often.
- **Replaced component.** The two cases show high counts of components that have been completely replaced. As in the case of new components, this stems from the fact that in both cases entire subsystems have been replaced on occasion. What this means is that even if it appears that replacing components is common, in fact it only happens rarely. An example of this category is the replacement of the processing-node in release 3 of the Billing Gateway that forced a replacement of all of the components inside it. Consequently, adding new component implementations, as described above, is a common evolution category, whereas replacing individual components singly and replacing both the framework interface and all of the implementations is much rarer.

7.4. Guidelines for software product-line evolution

We have evaluated the two cases with respect to the evolution categories, and discussed the categories we observed that occurred most frequently. Out of this analysis we present a set of seven guidelines focused on the technical aspects of software product-line evolution. These guidelines are useful when creating a new software product line, but are even more important if there already exist a product line that is evolving. Each of the guidelines present an area for future research, in order to help the evolution work by proposing methods and techniques to avoid the common pitfalls of software product-line development.

- **Avoid adjusting component interfaces.** We have found that sooner or later the interfaces of components will need to be changed because a new implementation requires more of

the interface than was planned initially. The trick is to delay this moment for as long as possible. There are two ways in which an interface can be altered—it can grow to provide more functionality, or it can shrink to provide less functionality. The former is the result of an external component requiring more of the component than it can currently cope with. To add functionality to a component is laborious but it is easy to find where the change needs to be implemented: in all the implementations of the component. There are also cases where the calling components need to be found and modified as well. A shrinking interface may be caused by the introduction of a new component that is specialized on a type of functionality previously handled by another component. This change will have an impact on the component implementations, in order to reduce the space required by the component and also to use the new component wherever the functionality is used internally. Furthermore, all external components that use the functionality will also need to be found and modified to use the new component. There may exist some ways to solve growing interfaces without touching the existing implementations. If the new functionality is well behaved it can be added as a small component next to the previous implementations, and connect to them using an existing interface. In most other cases there is no solution but to modify the original source code. We find this a very interesting area for future research—how to design components for growing interfaces without doing wrapping.

- **Focus on making component interfaces general.** During our studies, we have found that the prevalent form of evolution is that of adding new functionality to the existing components, and thus increasing the support of the industry standards. When designing the architecture and the component interfaces, a key focus is on how new implementations of functionality in components are added. Issues concerning this are where to keep functionality that at the design stage is perceived to be common to all implementations, and what actually to include in the interface. This guideline is highly related to the previous one, which discusses this issue under the evolution of the software product line, whereas this one is more focused on the initial design of the architecture. To the best of our knowledge, the way to find out if an interface is general enough is to make a thorough domain analysis. Bengtsson and Bosch (1999, 2000) present an alternative view of how to evaluate the architecture and the interfaces for evolvability, based on maintenance scenarios prepared by maintainers with more or less experience.
- **Separate domain and application behavior.** It is common that a component implements not only the domain functionality that it is supposed to support, but also a number of facilitating functionalities that are more focused on providing application structure and support. This can be things like reference counting for memory management or dependency graphs to start up the components in the correct sequence. The problem is that such functionality has a tendency to evolve more than the domain functionality, in that frequently new features are requested to provide even better control. If this functionality is embedded in the same component implementations as the domain functionality, not only must all component implementations be altered for the new control functionality, but the maintainers must also manually find their way between domain and control functionality to find out where to make the changes. To this end, we suggest that application functionality be separated from the domain functionality, and that aggregate relationships be used as the only connector between the two. This will not alleviate having to make changes to the application functionality, but it will help in making the

domain functionality more reusable, and less prone to require modifications is the application functionality changes.

- **Keep the software product line intact.** The term 'software product line' suggests that there is one product line per type of product, and if a company develops more than one type of product it also has more than one product line. This is normally not the case. In many cases, the term 'company software architecture' better depicts the reality. The term 'product' denotes, in such cases, only variations on a theme, whereas 'product family' is used to refer to the product areas in general. As an example, Axis Communications has a storage product family, a camera product family, and a scanner product family. The architectures in these systems are, however, very similar, and many assets can be reused in more than one product family.
  A new product is normally created by exchanging some component implementation; for example, Ethernet to Token Ring. With this view, different products are mostly created for marketing reasons. Technically, products are managed by configuration management, i.e., they have different make-files. Product families, on the other hand, are created by exchanging parts of the architecture, i.e., by replacing components. Unless there are some very compelling reasons; for example, that the new product family is written in a new language, one should strive to keep as much as possible from the other product families in order to create only a branch in the product line.
  This seems to be common sense, but there are often forces at work in a different direction. For example, some personnel may advocate a split-up of the software product line as well. Other personnel may advocate more change than the minimum really necessary, as for instance creating two versions: an old one, and a new one for the new product family. Often, however, even if a component does not provide all of the functionality one may wish for, it will be cheaper to enhance the existing component to supply the new functionality and adapt the previous products to this, than to maintain two parallel versions of the same component, with all that this brings in the form of bug fixes and synchronization headaches. Sooner or later, a decision usually is made to join the two versions anyhow.
- **Detect and exploit common functionality in component implementations.** Developing a new framework implementation is often done by finding the most similar implementation and writing the new code using the old one as a template. This leads to very similar code—code that could have been reused instead. We suggest that every domain component should have at least one library component in its tow, where common functionality between component implementations can be placed as soon as it is identified as shared between implementations. Development of a new implementation then also becomes a 'reuse inspection' for the implementation used as a template. The trouble with this approach arises when it is realized that the functionality put into the library component was in fact just common for two of the implementations, and that all the other implementations use another functionality. Also, the impact of making changes in the shared functionality may be hard to estimate.
- **Be open to rewriting components and implementations.** As we have shown, there are situations when a particular implementation, or maybe even an entire component, simply cannot cope with a desired change. Rather than forcing the change into the component, which results in patchy code and shortens its life even further, we suggest that the option to rewrite the component from scratch be considered. Even if the rewritten code does not get better than the original code, it will certainly be better than the code produced by patching the previous

implementation. Our two case studies suggest that it is common practice in industry to rewrite components like this.

- **Avoid hidden assumptions and design decisions in the source code.** Twice in our study we have encountered situations in which an entire component, or even a set of components, has been discarded and rewritten from scratch. The reason for this has been that some design decisions do not take on an explicit representation in the architecture, and become hidden implicitly in the source code. When such assumptions or design decisions change, it is usually easier to rewrite the entire component, rather than to change the assumptions implicit in the existing code. Our solution proposal to this is to try to give all design decisions an explicit representation and actualisation in the form of components or design patterns, so that the effects of changing these decisions are minimized.

Acknowledgements

We thank the two Swedish companies involved in the two case studies, Axis Communications AB in Lund, and Ericsson Software Technology in Ronneby. We also thank our colleagues Per Olof Bengtsson and Michael Mattsson for valuable comments and encouragement.

References

Bansiya J. 1999a. Assessing maturity of object-oriented application frameworks. In *Object-Oriented Application Frameworks; Problems & Perspectives*, Fayed ME, Schmidt DC and Johnson RE (eds). John Wiley & Sons Inc.: New York, NY; forthcoming.

Bansiya J. 1999b. Assessment of application framework maturity using design metrics http://indus.ca.uah.edu/research-papers.htm [6 September 1999]

Basili V, Briand L, Condon S, Kim YM, Melo WL, Valett J. 1996. Understanding and predicting the process of software maintenance releases. In *Proceedings 18th International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos CA; pp. 464–474.

Bass L, Clements P, Cohen S, Northrop L, Withey J. 1997. *Product Line Practice Workshop Report, CMU/SEI-97-TR-003*. Software Engineering Institute, Carnegie Mellon University: Pittsburgh PA.

Bass L, Clements P, Kazman R. 1998. *Software Architecture in Practice*. Addison-Wesley Publishing Co.: Reading MA.

Bass L, Chastek G, Clements P, Northrop L, Smith D, Withey J. 1998. *Second Product Line Practice Workshop Report, CMU/SEI-98-TR-015*. Software Engineering Institute, Carnegie Mellon University: Pittsburgh PA.

Bengtsson PO, Bosch J. 1999. Architecture level prediction of software maintenance. In *Proceedings 3rd European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press: Los Alamitos CA; pp. 139–147.

Bengtsson PO, Bosch J. 2000. An experiment on creating scenario profiles for software change. In *Annals of Software Engineering*, Cifuentes C and Bailes P (eds). Baltzer Science Publishers: Bussum, The Netherlands; forthcoming.

Bosch J. 1999a. Evolution and composition of reusable assets in product-line architectures: a case study. In *Software Architecture*, Donohoe P (ed). Kluwer Academic Publishers: Boston MA; pp. 321–339.

Bosch J. 1999b. Product-line architectures in industry: a case study. In *Proceedings 21st International Conference on Software Engineering*. ACM Press: New York NY; pp. 544–554.

Brooks FP. 1995. *The Mythical Man Month*. Addison-Wesley Publishing Co.: Reading MA.

Chapin N. 1988. Software maintenance life cycle. In *Proceedings Conference on Software Maintenance–1988*. IEEE Computer Society Press: Los Alamitos CA; pp. 6–13.

Dikel D, Kane D, Ornburn S, Loftus W, Wilson J. 1997. Applying software product-line architecture. *IEEE Computer* **30**(8):49–55.

Gall H, Jazayeri M, Klösch RG, Trausmuth G. 1997. Software evolution observations based on product release history. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA; pp. 160–166.

Gamma E, Helm R, Johnson R, Vlissides J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co.: Reading MA.

Griss ML, Favaro J, d'Alessandro M. 1998. Integrating feature modeling with the RSEB. In *Proceedings 5th International Conference on Software Reuse*. IEEE Computer Society: Los Alamitos CA; pp. 76–85.

Jacobson I, Griss M, Johnson P. 1997. *Software Reuse: Architecture, Process, and Organization for Business Success*. Addison-Wesley Publishing Co.: Reading MA.

Kang K, Cohen S, Heiss J, Novak W, Peterson A, 1990. *Feature-Oriented Domain Analysis Feasibility Study, SEI Technical Report CMU/SEI-90-TR-21*. Software Engineering Institute, Carnegie Mellon University: Pittsburgh PA.

Lehman MM. 1980. On understanding laws, evolution and conservation in the large program life cycle. *J. Systems Software* **1**(3):213–221.

Lehman MM. 1994. Software evolution. In *Encyclopedia of Software Engineering*, Marciniak JL (ed). John Wiley & Sons Inc.: New York NY; pp. 1202–1208.

Linden vdF (ed). 1998. Development and evolution of software architectures for product families. In *Proceedings 2nd International ESPRIT ARES Workshop (Lecture Notes in Computer Science 1429)*. Springer-Verlag: Berlin.

Lindvall M, Runesson M, 1998. The visibility of maintenance in object models: an empirical study. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA; pp. 54–62.

Macala RR, Stuckey LD, Gross DC. 1996. Managing domain-specific, product-line development. *IEEE Software* **13**(3):57–67.

Mattsson M, Bosch J, 1998. Frameworks as components: a classification of framework evolution. In *Proceedings Nordic Workshop on Programming Environment Research 1998*. Bergin Print Service: Bergen Norway; pp. 163–174.

Mattsson M, Bosch J, 1999a. Evolution observations of an industry object-oriented framework. In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA; pp. 139–145.

Mattsson M, Bosch J, 1999b. Characterizing stability in evolving frameworks. In *Proceedings TOOLS Europe 1999*. IEEE Computer Society Press: Los Alamitos CA; pp. 118–130.

Mattsson M, Bosch J. 1999c. Assessing maturity of object-oriented application frameworks. In *Object-Oriented Application Frameworks; Problems & Perspectives*, Fayed ME, Schmidt DC and Johnson RE (eds). John Wiley & Sons Inc.: New York, NY; forthcoming.

McCall JA. 1994. Quality factors. In *Encyclopedia of Software Engineering*, Marciniak JL (ed). John Wiley & Sons Inc.: New York, NY; pp. 958–969.

Mcllroy MD. 1969. Mass produced software components. In *Software Engineering: Concepts and Techniques*, Buxton JM, Naur P and Randell B (eds). Petrocelli/Charter Publishers Inc. (republished 1976): New York, NY; pp. 88–98.

Parnas DL. 1976. On the design and development of program families. *IEEE Transactions on Software Engineering* **SE-2**(1):1–9.

Perry DE, Siy HP. 1998. Parallel changes in large scale software development: an observational case study. In *Proceedings 20th International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos CA; pp. 251–260.

Roberts D, Johnson RE. 1996. Evolving frameworks: a pattern language for developing object-oriented frameworks. In *Pattern Languages of Programming Design 3*, Martin R, Riehle D and Buschmann F (eds). Addison-Wesley Publishing Co.: Reading MA; pp. 471–486.

Shaw M, Garlan D. 1996. *Software Architecture—Perspectives on an Emerging Discipline*. Prentice-Hall: Saddle River NJ.

Tracz W. 1995. DSSA (domain-specific software architecture): pedagogical example. *Software Engineering Notes* **20**(3):49–62.

Authors' biographies:

Mikael Svahnberg is a Ph.D. candidate at the University of Karlskrona/Ronneby in Sweden. He received an M.Sc. degree from the same university in 1998. His research activities include software product lines, object-oriented frameworks, and the evolution of software product lines. His email address is: Mikael.Svahnberg@ipd.hk-r.se

Jan Bosch is a Professor of Software Engineering at the University of Karlskrona/Ronneby in Sweden, where he heads the RISE (Research In Software Engineering research group). He received an M.Sc. degree from the University of Twente in The Netherlands, and a Ph.D. degree from Lund University in Sweden. His research activities include software architecture design, product-line architectures, object-oriented frameworks and component-oriented programming. He was the co-editor of the ECOOP'97 and ECOOP'98 workshop readers published by Springer-Verlag in the LNCS series, and the initiator and coordinator of NOSAR, a Nordic network on software architecture involving both academia and industry. His email address is: Jan.Bosch@ipd.hk-r.se

